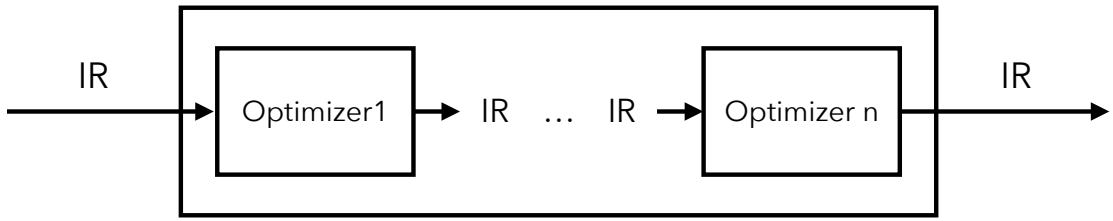
COSE312: Compilers

Lecture 16 — Optimization

Hakjoo Oh
2025 Spring

Middle End: Optimizer

Converts the source program into a more efficient yet semantically equivalent program.



ex)

```
t1 = 10
t2 = rate * t1
t3 = init + t2
pos = t3
```

original IR

```
t1 = 10
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
pos = init + t2
```

final IR

Common Optimization Passes

- Common subexpressions elimination
- Copy propagation
- Deadcode elimination
- Constant folding
- ...

Common Subexpression Elimination

- An occurrence of an expression $\boldsymbol{E}$ is called a *common subexpression* if $\boldsymbol{E}$ was previously computed and the values of the variables in $\boldsymbol{E}$ have not changed since the previous computation.

```
x = 2 * k + 1
...              // no defs to k
y = 2 * k + 1
```

- We can avoid recomputing $\boldsymbol{E}$ by replacing $\boldsymbol{E}$ by the variable that holds the previous value of $\boldsymbol{E}$.

```
x = 2 * k + 1
...              // no defs to k
y = x
```

Copy Propagation

After the copy statement $u = v$, use v for u unless u is re-defined.

```
u = v                          u = v
x = u + 1                      x = v + 1
u = x              =>          u = x
y = u + 2                      y = x + 2
```

cf) Copy instructions can be generated during optimization, e.g., common subexpression elimination.

Deadcode Elimination

- A variable is *live* at a point in a program if its value is used eventually; otherwise it is *dead* at that point.
- A statement is said to be *deadcode* if it computes values that never get used.

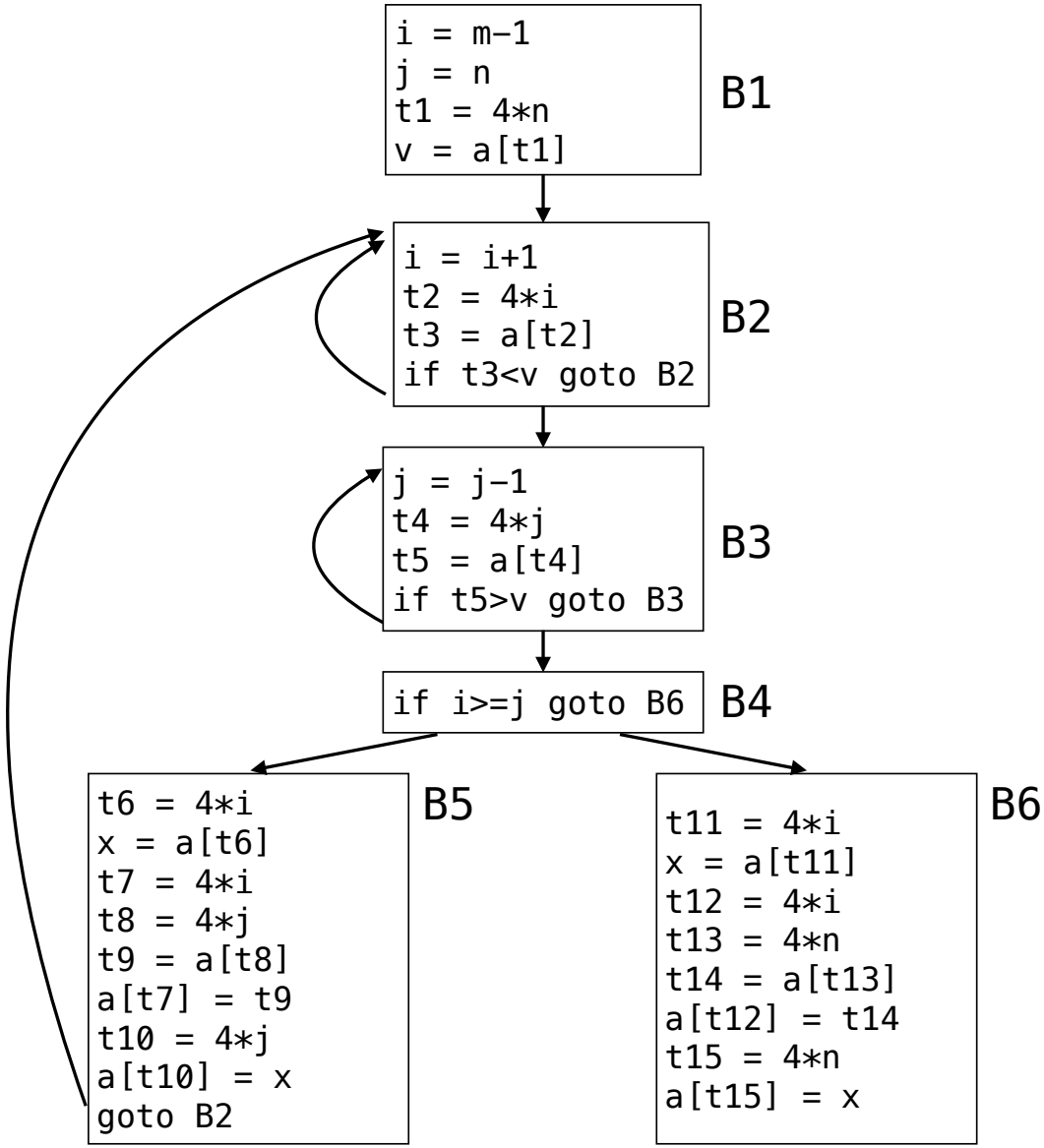
```
u = v      // deadcode
x = v + 1
u = x.     // deadcode
y = x + 2
```

Constant Folding

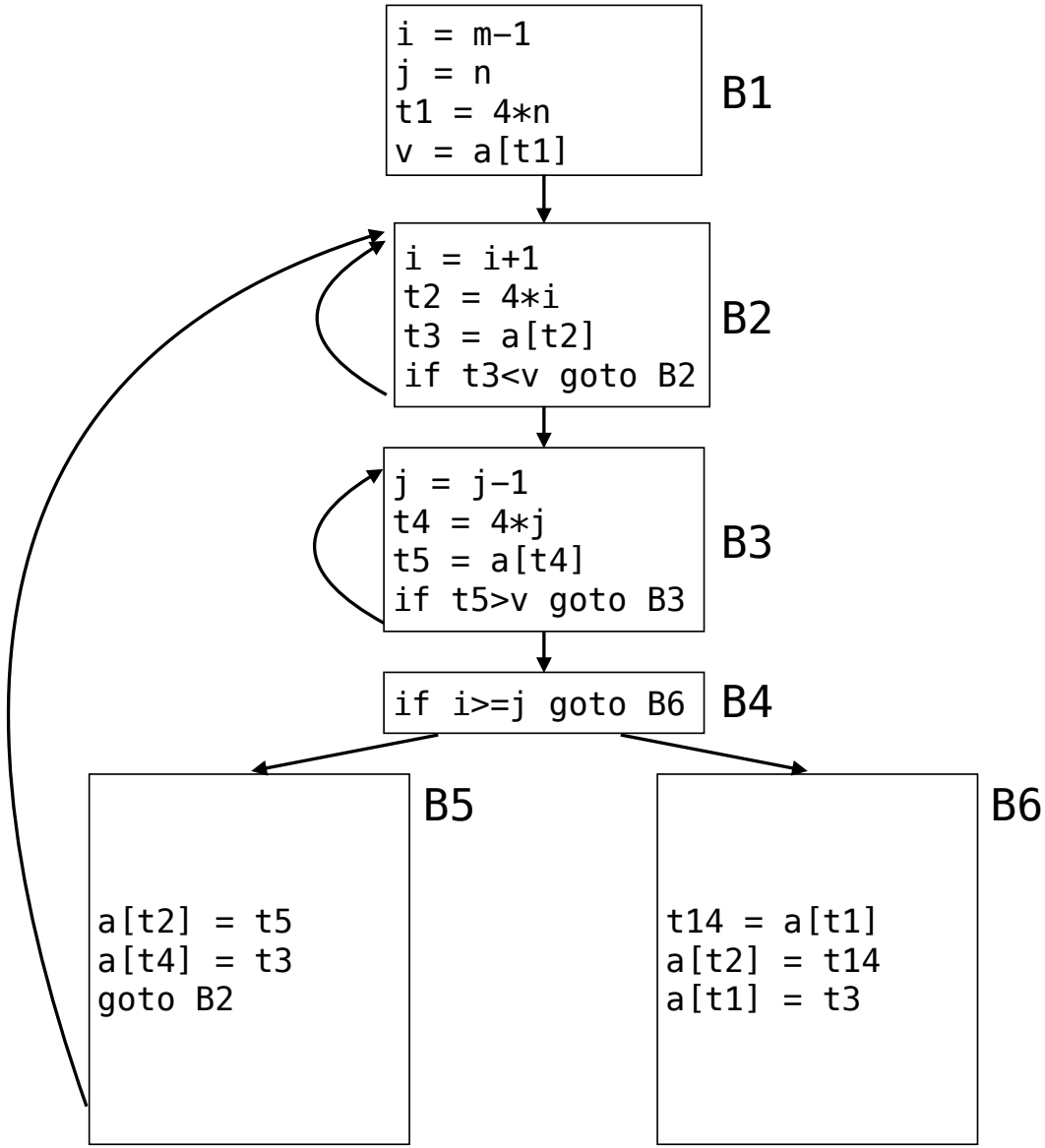
Decide that the value of an expression is a constant and use the constant instead.

```
c = 1                          c = 1
x = c + c          =>          x = 2
y = x + x                      y = 4
```

Example: Original Program



Example: Optimized Program



Static analysis is needed

To optimize a program, we need static analysis that derives information about the flow of data along program execution paths. Examples:

- Do the two textually identical expressions evaluate to the same value along any possible execution path of the program? (If so, we can apply common subexpression elimination)
- Is the result of an assignment not used along any subsequent execution path? (If so, we can apply deadcode elimination).

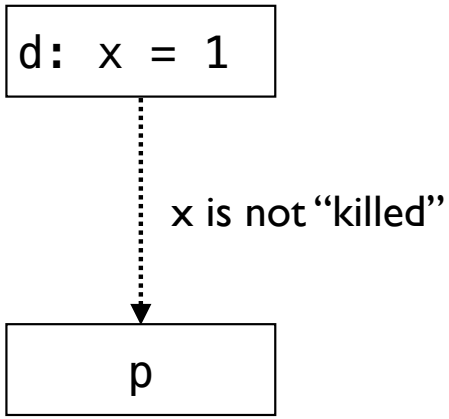
Data-Flow Analysis

A collection of program analysis techniques that derive information about the flow of data along program execution paths, enabling safe code optimization, bug detection, etc.

- Reaching definitions analysis
- Live variables analysis
- Available expressions analysis
- Constant propagation analysis
- ...

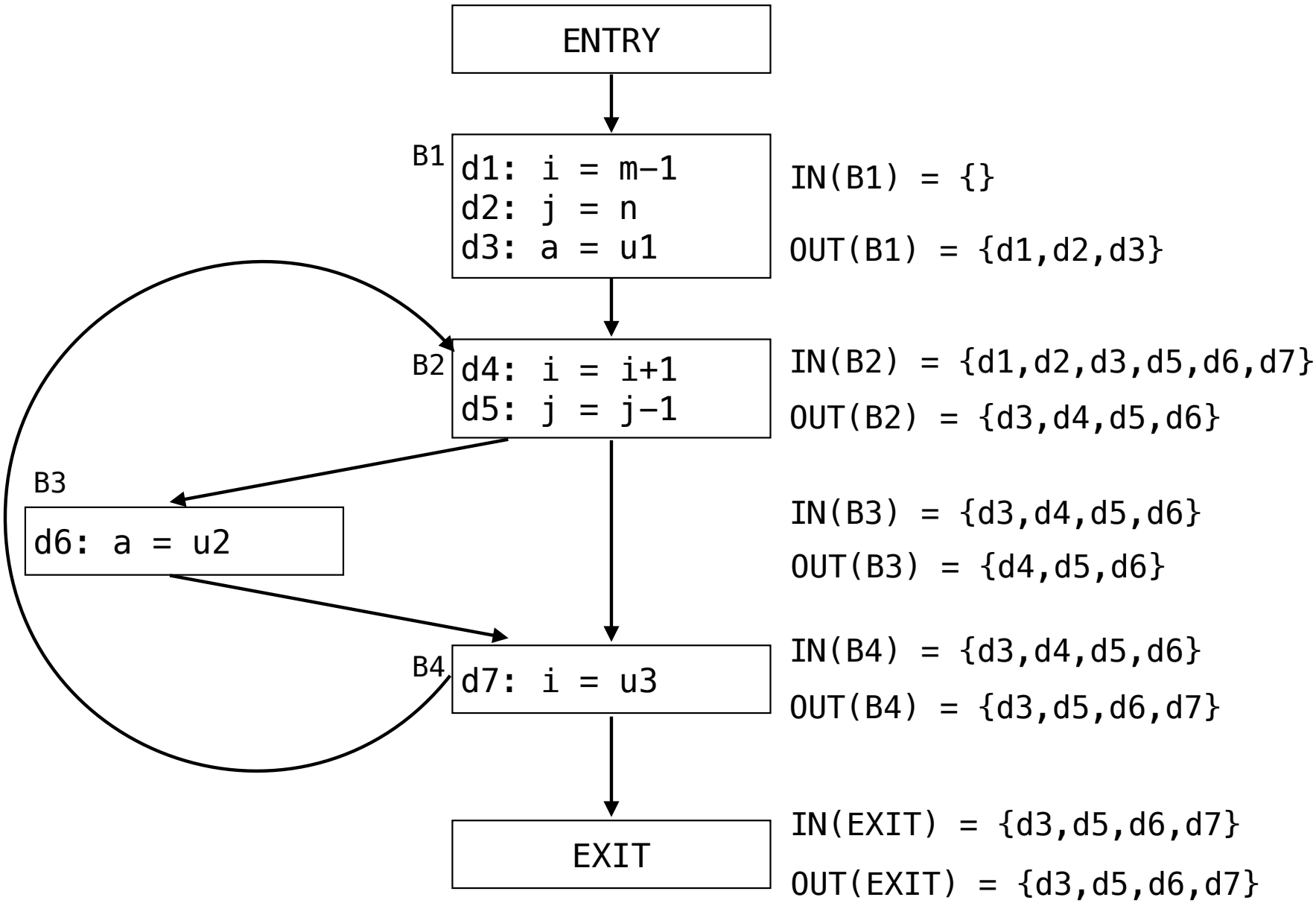
Reaching Definitions Analysis

- A definition d *reaches* a point p if there is a path from the definition point to p such that d is not "killed" along that path.



- For each program point, RDA finds definitions that *can* reach the program point along some execution paths.

Example: Reaching Definitions Analysis



Applications

Reaching definitions analysis has many applications, e.g.,

- Simple constant propagation
  - For a use of variable v in statement n: $\boxed{n : x = ...v...}$
  - If the definitions of v that reach n are all of the form $\boxed{d : v = c}$
  - Replace the use of v in n by c
- Uninitialized variable detection
  - Put a definition $\boxed{\text{d: x = any}}$ at the program entry.
  - For a use of variable x in statement n: $\boxed{n : v = ...x...}$
  - If d reaches n, x is potentially uninitialized.
  - ```
    ...
    if (...) x = 1;
    ...
    a = x
    ```
- Loop optimization
  - If all of the reaching definitions of the operands of n are outside of the loop, then n can be moved out of the loop ("loop-invariant code motion")
  - `while (...) {...; n: z = x + y; ... }`
    ```

The Analysis is Conservative

- Exact reaching definitions information cannot be obtained at compile time. It can be obtained only at runtime.
- ex) Deciding whether each path can be taken is undecidable:

```
a = rand(); b = rand(); c = rand(); k = rand();
if (a^k + b^k != c^k) {     // always true
  // (1)
} else {
  // (2)
}
```

- RDA computes an over-approximation of the reaching definitions that can be obtained at runtime.

Reaching Definitions Analysis

The goal is to compute

$$\begin{aligned} \textbf{in} &: \mathit{Block} \to 2^{\mathit{Definitions}} \\ \textbf{out} &: \mathit{Block} \to 2^{\mathit{Definitions}} \end{aligned}$$

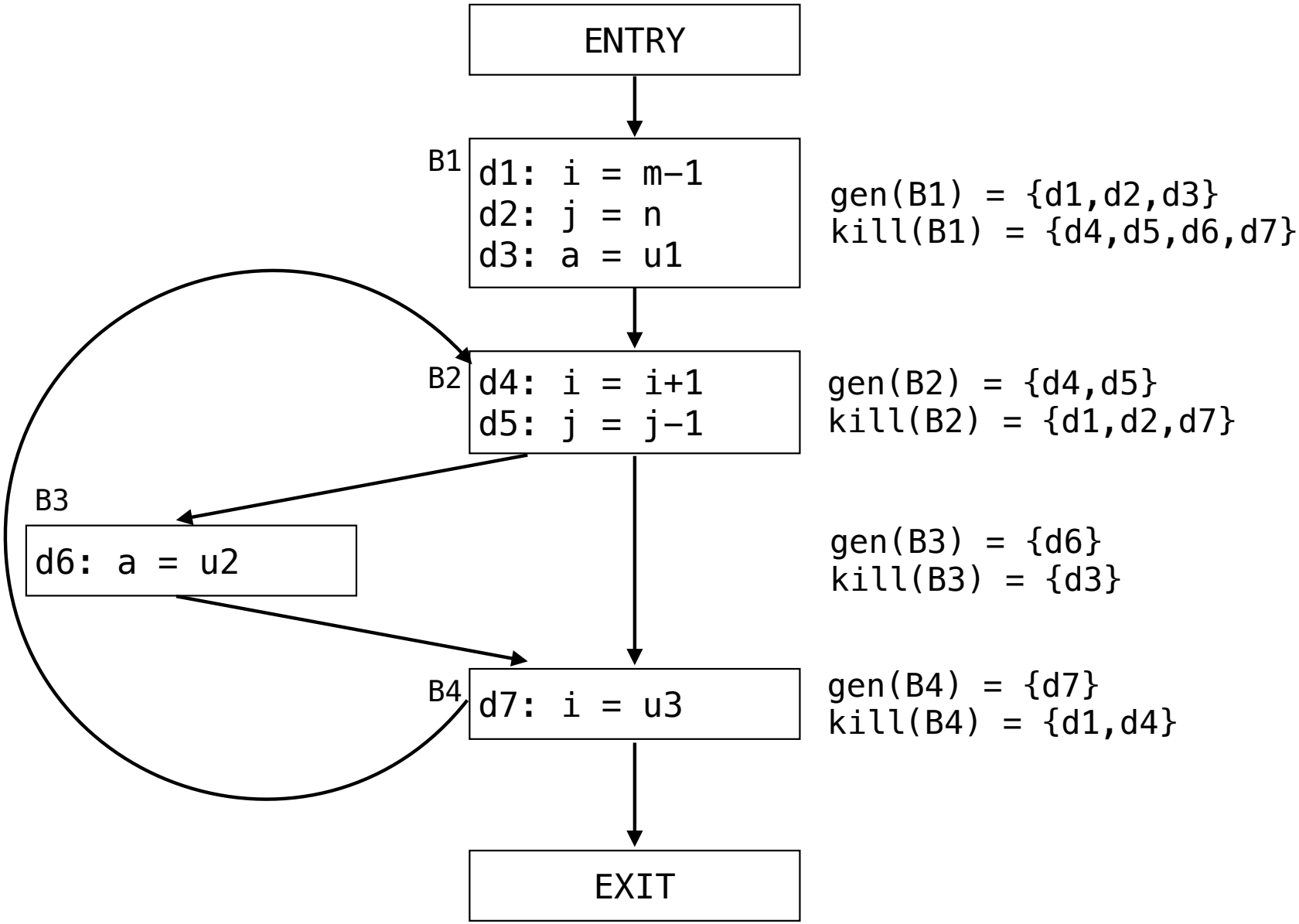
1. Compute gen/kill sets.
2. Derive transfer functions for each block in terms of gen/kill sets.
3. Derive the set of data-flow equations.
4. Solve the equation by the iterative fixed point algorithm.

1. Compute Gen/Kill Sets

$$\begin{aligned} \textbf{gen} &: \mathit{Block} \to 2^{\mathit{Definitions}} \\ \textbf{kill} &: \mathit{Block} \to 2^{\mathit{Definitions}} \end{aligned}$$

- **gen**(B): the set of definitions "generated" at block B
- **kill**(B): the set of definitions "killed" at block B

Example


ENTRY
B1
d1: i = m-1
d2: j = n
d3: a = u1
gen(B1) = {d1,d2,d3}
kill(B1) = {d4,d5,d6,d7}
B2
d4: i = i+1
d5: j = j-1
gen(B2) = {d4,d5}
kill(B2) = {d1,d2,d7}
B3
d6: a = u2
gen(B3) = {d6}
kill(B3) = {d3}
B4
d7: i = u3
gen(B4) = {d7}
kill(B4) = {d1,d4}
EXIT

Exercise

Compute the **gen** and **kill** sets for the basic block B:

```
d1: a = 3
d2: a = 4
```

- $\textbf{gen}(B) = \{d_2\}$
- $\textbf{kill}(B) = \{d_1, d_2, \ldots\}$

In general, when we have k definitions in a block B:

```
d1; d2; ...; d_k
```

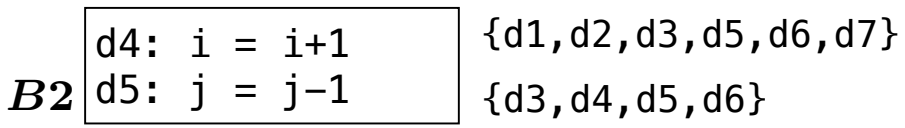
- $\textbf{gen}(B) =$
  $\textbf{gen}(d_k) \cup (\textbf{gen}(d_{k-1}) - \textbf{kill}(d_k)) \cup (\textbf{gen}(d_{k-2}) - \textbf{kill}(d_{k-1}) - \textbf{kill}(d_k)) \cup \cdots \cup (\textbf{gen}(d_1) - \textbf{kill}(d_2) - \textbf{kill}(d_3) - \cdots - \textbf{kill}(d_k))$
- $\textbf{kill}(B) = \textbf{kill}(d_1) \cup \textbf{kill}(d_2) \cup \cdots \cup \textbf{kill}(d_k)$

2. Transfer Functions

- The transfer function is defined for each basic block B:

$$f_B : 2^{Definitions} \rightarrow 2^{Definitions}$$

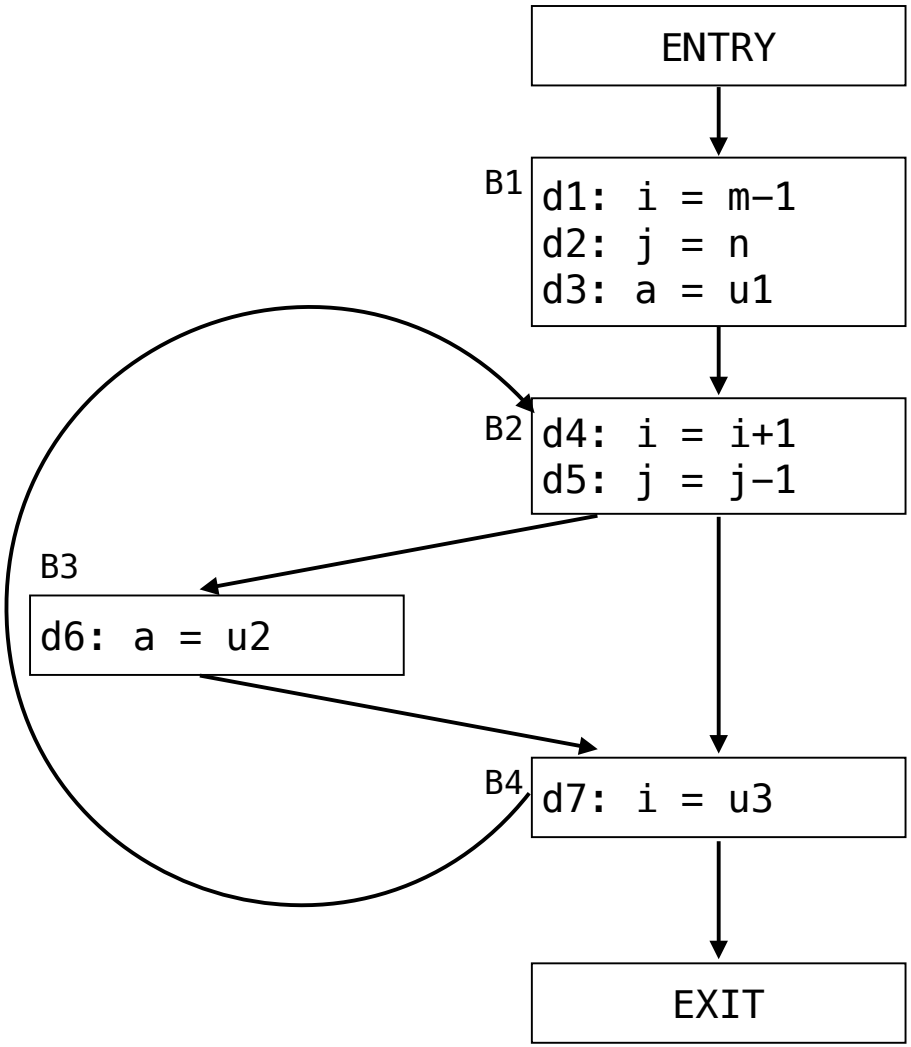
- The transfer function for a block B encodes the semantics of the block B, i.e., how the block transfers the input to the output.



- The semantics of B is defined in terms of **gen**(B) and **kill**(B):

$$f_B(X) = \textbf{gen}(X) \cup (X - \textbf{kill}(X))$$

3. Derive Data-Flow Equations



$$
\begin{aligned}
\mathbf{in}(B_1) &= \emptyset \\
\mathbf{out}(B_1) &= f_{B_1}(\mathbf{in}(B_1)) \\
\\
\mathbf{in}(B_2) &= \mathbf{out}(B_1) \cup \mathbf{out}(B_4) \\
\mathbf{out}(B_2) &= f_{B_2}(\mathbf{in}(B_2)) \\
\\
\mathbf{in}(B_3) &= \mathbf{out}(B_2) \\
\mathbf{out}(B_3) &= f_{B_3}(\mathbf{in}(B_3)) \\
\\
\mathbf{in}(B_4) &= \mathbf{out}(B_2) \cup \mathbf{out}(B_3) \\
\mathbf{out}(B_4) &= f_{B_4}(\mathbf{in}(B_4))
\end{aligned}
$$

Data-Flow Equations

In general, the data-flow equations can be written as follows:

$$\begin{aligned}
\textbf{in}(B_i) &= \bigcup_{P \hookrightarrow B_i} \textbf{out}(P) \\
\textbf{out}(B_i) &= f_{B_i}(\textbf{in}(B_i)) \\
&= \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i))
\end{aligned}$$

where $(\hookrightarrow)$ is the control-flow relation.

4. Solve the Equations

- The desired solution is the *least* **in** and **out** that satisfies the equations (why least?):

$$\begin{array}{rcl} \textbf{in}(B_i) & = & \bigcup_{P \hookrightarrow B_i} \textbf{out}(P) \\ \textbf{out}(B_i) & = & \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i)) \end{array}$$

- The solution is defined as $\boldsymbol{fix F}$, where $\boldsymbol{F}$ is defined as follows:

$$\boldsymbol{F} : (\boldsymbol{Block} \to \boldsymbol{2}^{\boldsymbol{Definitions}})^{\boldsymbol{2}} \to (\boldsymbol{Block} \to \boldsymbol{2}^{\boldsymbol{Definitions}})^{\boldsymbol{2}}$$

$$\boldsymbol{F}(\textbf{in}, \textbf{out}) = (\boldsymbol{\lambda B.} \bigcup_{\boldsymbol{P \hookrightarrow B}} \textbf{out}(P), \boldsymbol{\lambda B.} f_B(\textbf{in}(B))$$

The least fixed point $\boldsymbol{fix F}$ is computed by

$$\bigcup_{\boldsymbol{i \geq 0}} \boldsymbol{F}^i(\boldsymbol{\lambda B.\emptyset}, \boldsymbol{\lambda B.\emptyset})$$

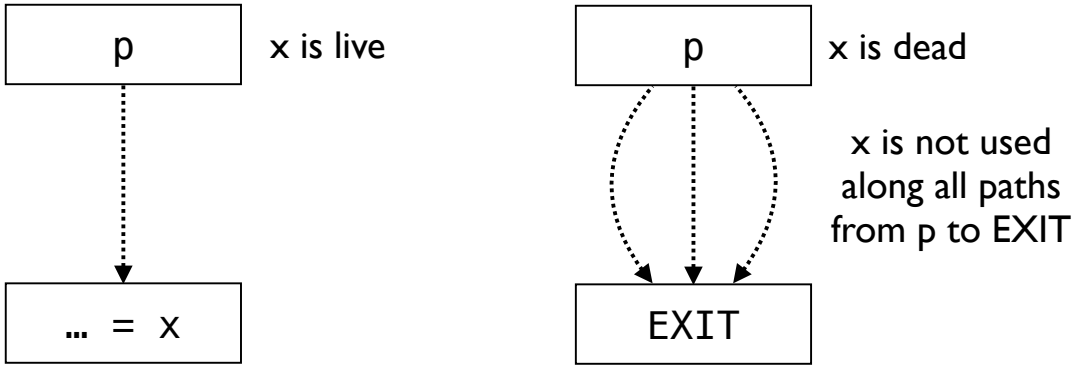
Fixed Point Algorithm

The equations are solved by the iterative fixed point algorithm:

For all i, $\textbf{in}(B_i) = \textbf{out}(B_i) = \emptyset$
while (changes to any **in** and **out** occur) {
  For all i, update
    $\textbf{in}(B_i) = \bigcup_{P \hookrightarrow B_i} \textbf{out}(P)$
    $\textbf{out}(B_i) = \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i))$
}

Liveness Analysis

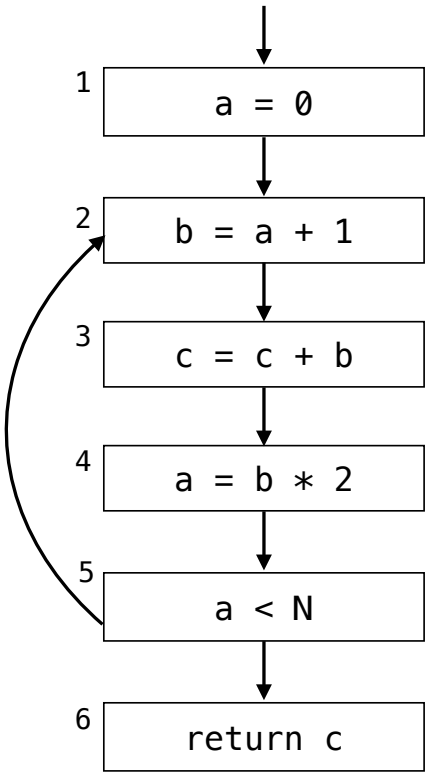
- A variable is *live* at program point p if its value could be used in the future (along some path starting at p).



- Liveness analysis aims to compute the set of live variables for each basic block of the program.

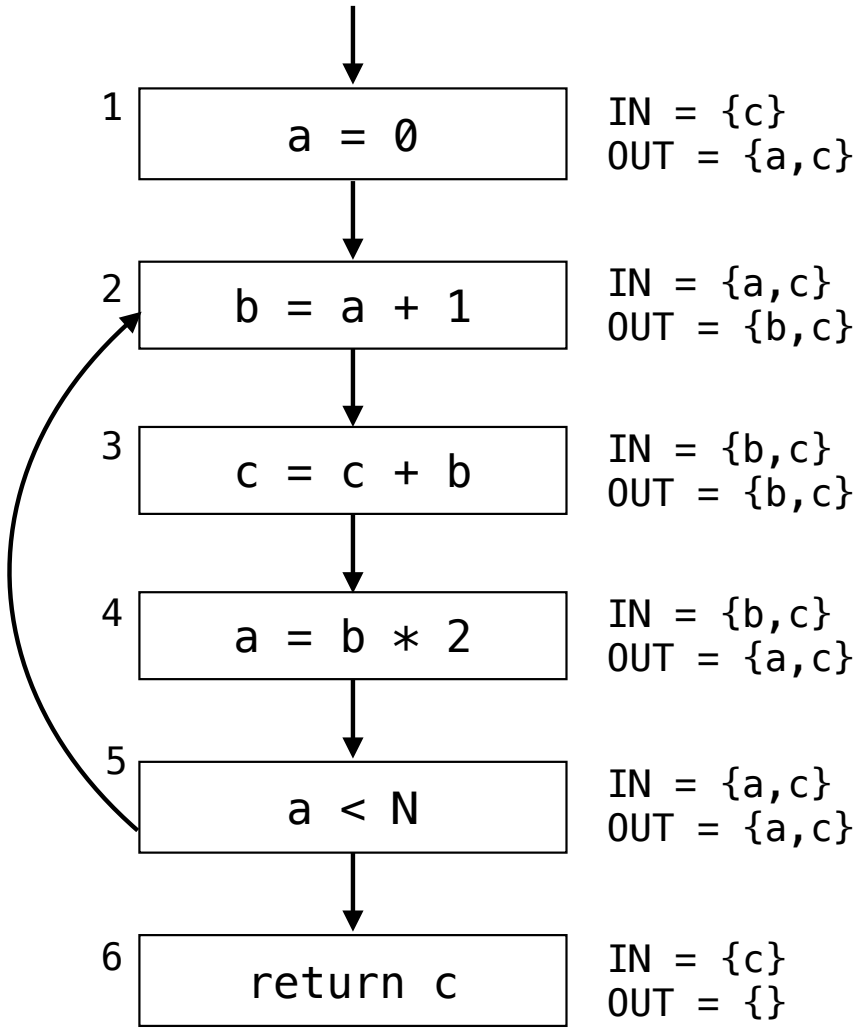
Example: Liveness of Variables

We analyze liveness from the future to the past.



- The live range of b: $\{2 \to 3, 3 \to 4\}$
- The live range of a: $\{1 \to 2, 4 \to 5 \to 2\}$ (not from $2 \to 3 \to 4$)
- The live range of c: the entire code

Example: Liveness of Variables



Applications

- Deadcode elimination
  - Problem: Eliminate assignments whose computed values never get used.
  - Solution: How?
  - Suppose we have a statement: $\boxed{\text{n: x} = \text{y} + \text{z}}$
  - When x is dead at n, we can eliminate n.
- Uninitialized variable detection
  - Problem: Detect uninitialized use of variables
  - Solution: How? Any variables live at the program entry (except for parameters) are potentially uninitialized
- Register allocation
  - Problem: Rewrite the intermediate code to use no more temporaries than there are machine registers
  - Example:

```
a := c + d                r1 := r2 + r3
e := a + b                r1 := r1 + r4
f := e - 1                r1 := r1 - 1
```

  - Solution: How? Compute live ranges of variables. If two variables a and b never live at the same time, assign the same register to them.

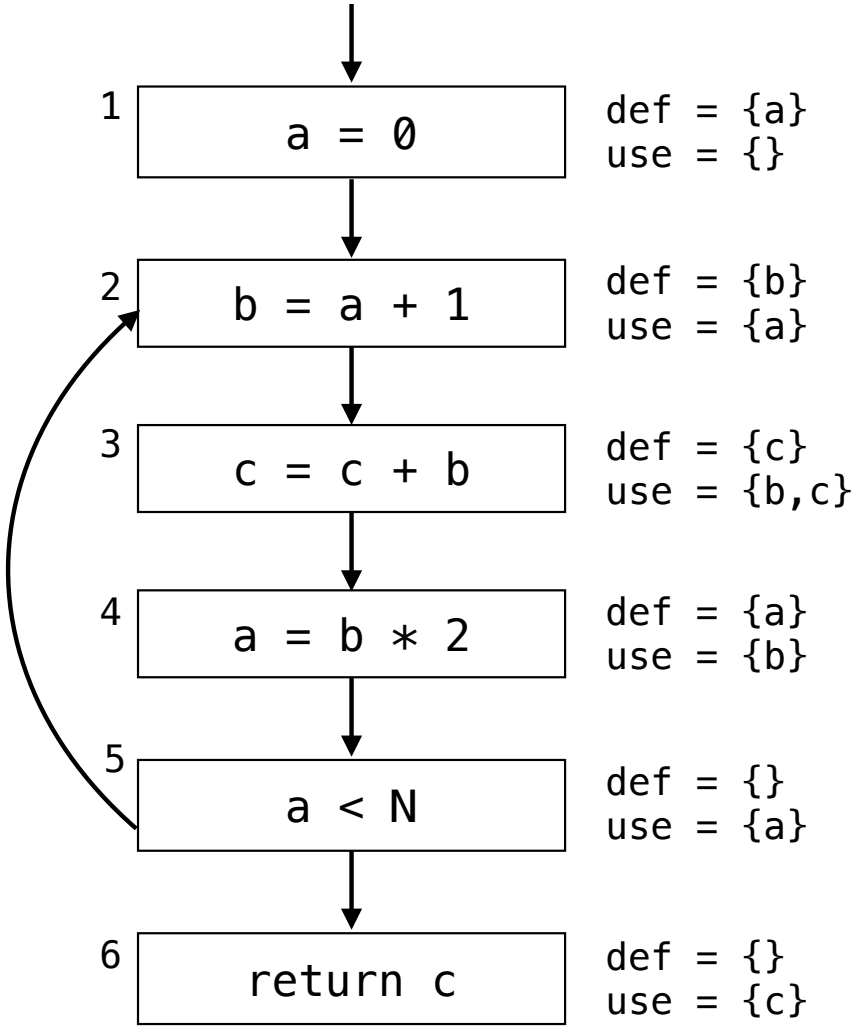
Liveness Analysis

The goal is to compute

$$\begin{aligned} \textbf{in} &: \mathit{Block} \to 2^{\mathit{Var}} \\ \textbf{out} &: \mathit{Block} \to 2^{\mathit{Var}} \end{aligned}$$

1. Compute def/use sets.
2. Derive transfer functions for each basic block in terms of def/use sets.
3. Derive the set of data-flow equations.
4. Solve the equation by the iterative fixed point algorithm.

Def/Use Sets


1
a = 0
def = {a}
use = {}
2
b = a + 1
def = {b}
use = {a}
3
c = c + b
def = {c}
use = {b,c}
4
a = b * 2
def = {a}
use = {b}
5
a < N
def = {}
use = {a}
6
return c
def = {}
use = {c}

cf) Def/Use sets are only dynamically computable

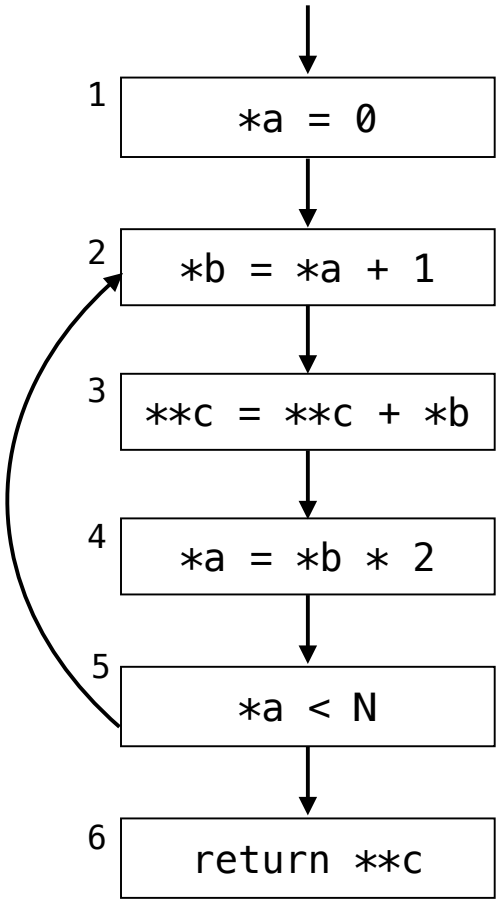
In general, we need *pointer analysis* to compute (may/must) def/use sets.

may points-to information

```
a ↦ { x, y }
b ↦ { z }
c ↦ { a }
```

must points-to information

```
a ↦ { x }
b ↦ { z }
c ↦ { a }
```



	may	must
1	def = { x, y } use = { a }	def = { x } use = { a }
2	def = { z } use = { a, b, x, y }	def = { z } use = { a, b, x }
3	def = { x, y } use = { b, c, x, y, z }	def = { x } use = { b, c, x, z }
4	def = { x, y } use = { a, b, z }	def = { x } use = { a, b, z }
5	def = {} use = { a, x, y }	def = {} use = { a, x }
6	def = {} use = { c, x, y }	def = {} use = { c, x }

Data-Flow Equations

Intuitions:

1. If a variable is in $\mathbf{use}(B)$, then it is live on entry to block B. (Assume B has a single statement for simplicity)
2. If a variable is live at the end of block B, and not in $\mathbf{def}(B)$, then the variable is also live on entry to B.
3. If a variable is live on enty to block B, then it is live at the end of predecessors of B.

Equations:

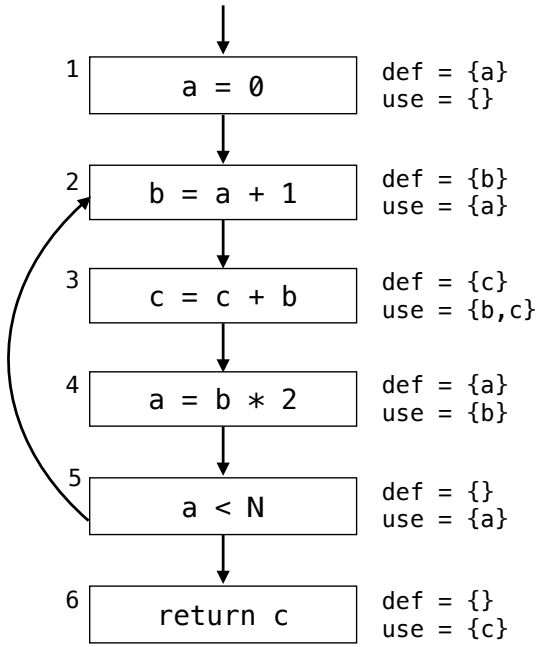
$$\mathbf{in}(B) = \mathbf{use}(B) \cup (\mathbf{out}(B) - \mathbf{def}(B))$$

$$\mathbf{out}(B) = \bigcup_{B \hookrightarrow S} \mathbf{in}(S)$$

Fixed Point Computation

For all i, $\mathbf{in}(B_i) = \mathbf{out}(B_i) = \emptyset$
while (changes to any **in** and **out** occur) {
  For all i, update
    $\mathbf{in}(B_i) = \mathbf{use}(B_i) \cup (\mathbf{out}(B_i) - \mathbf{def}(B_i))$
    $\mathbf{out}(B_i) = \bigcup_{B_i \hookrightarrow S} \mathbf{in}(S)$
}

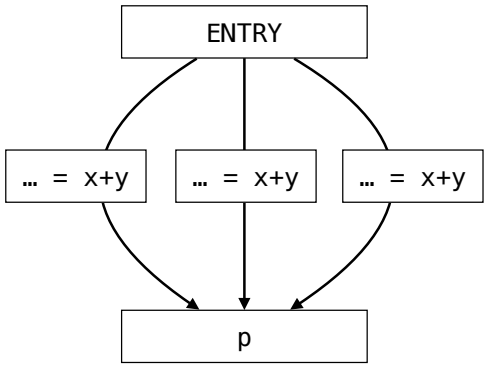
Example



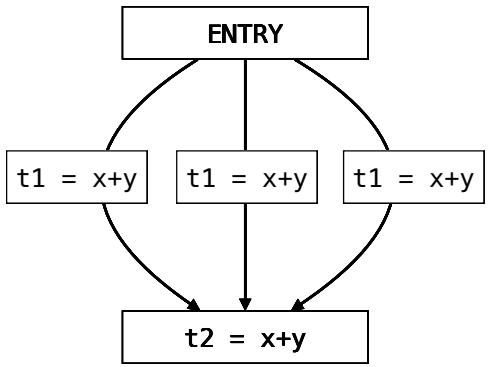
			$1st$		$2nd$		$3rd$	
	use	def	out	in	out	in	out	in
6	$\{c\}$	$\emptyset$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$
5	$\{a\}$	$\emptyset$	$\{c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$
4	$\{b\}$	$\{a\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$
3	$\{b,c\}$	$\{c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$
2	$\{a\}$	$\{b\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$
1	$\emptyset$	$\{a\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$

Available Expressions Analysis

- An expression $x + y$ is *available* at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.



- Application: common subexpression elimination (i.e., given a program that computes e more than once, eliminate one of the duplicate computations)



Available Expressions Analysis

The goal is to compute

$$\begin{aligned} \textbf{in} &: \mathit{Block} \to 2^{\mathit{Expr}} \\ \textbf{out} &: \mathit{Block} \to 2^{\mathit{Expr}} \end{aligned}$$

1. Derive the set of data-flow equations.
2. Solve the equation by the iterative fixed point algorithm.

Gen/Kill Sets

- **gen**(B): the set of expressions evaluated and not subsequently killed
- **kill**(B): the set of expressions whose variables can be killed
- What expressions are generated and killed by each of statements?

Statement s	**gen**(s)	**kill**(s)
$x = y + z$	$\{y + z\} - \textbf{kill}(s)$	expressions containing x
$x = \texttt{alloc}(n)$	$\emptyset$	expressions containing x
$x = y[i]$	$\{y[i]\} - \textbf{kill}(s)$	expressions containing x
$x[i] = y$	$\emptyset$	expressions of the form $x[k]$

($x = y + z$ generates $y + z$, but $y = y + z$ does not because y is subsequently killed.)

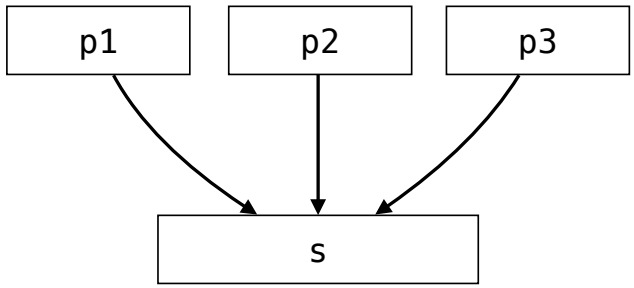
- What expressions are generated and killed by the block?

$$\begin{array}{|l|}\hline a = b + c \\ b = a - d \\ c = b + c \\ d = a - d \\ \hline\end{array}$$

1. Set up a set of data-flow equations

Intuitions:

1. At the entry, no expressions are available.
2. An expression is available at the entry of a block only if it is available at the end of *all* its predecessors.



Equations:

$$\mathbf{in}(ENTRY) = \emptyset$$

$$\mathbf{out}(B) = \mathbf{gen}(B) \cup (\mathbf{in}(B) - \mathbf{kill}(B))$$

$$\mathbf{in}(B) = \bigcap_{P \to B} \mathbf{out}(B)$$

2. Solve the equations

- We are interested in the largest set satisfying the equation
- Need to find the greatest solution (i.e., greatest fixed point) of the equation.

$$\textbf{in}(\boldsymbol{ENTRY}) = \emptyset$$

For other $\boldsymbol{B_i}$, $\textbf{in}(\boldsymbol{B_i}) = \textbf{out}(\boldsymbol{B_i}) = \boldsymbol{Expr}$

while (changes to any **in** and **out** occur) {

For all $\boldsymbol{i}$, update

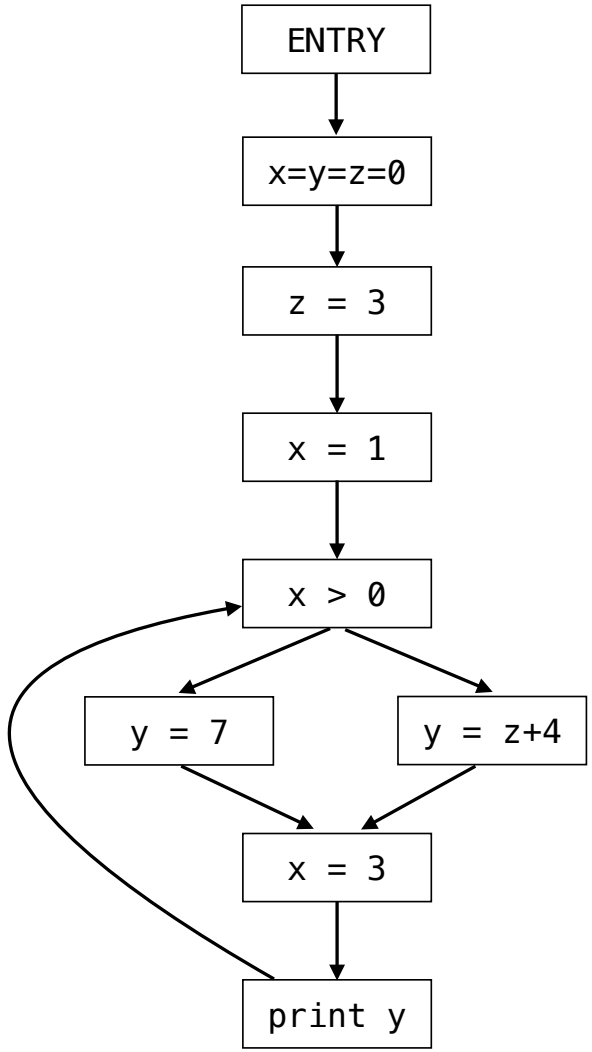
$$\textbf{in}(\boldsymbol{B_i}) = \bigcap_{\boldsymbol{P \hookrightarrow B_i}} \textbf{out}(\boldsymbol{P})$$

$$\textbf{out}(\boldsymbol{B_i}) = \textbf{gen}(\boldsymbol{B_i}) \cup (\textbf{in}(\boldsymbol{B_i}) - \textbf{kill}(\boldsymbol{B_i}))$$

}

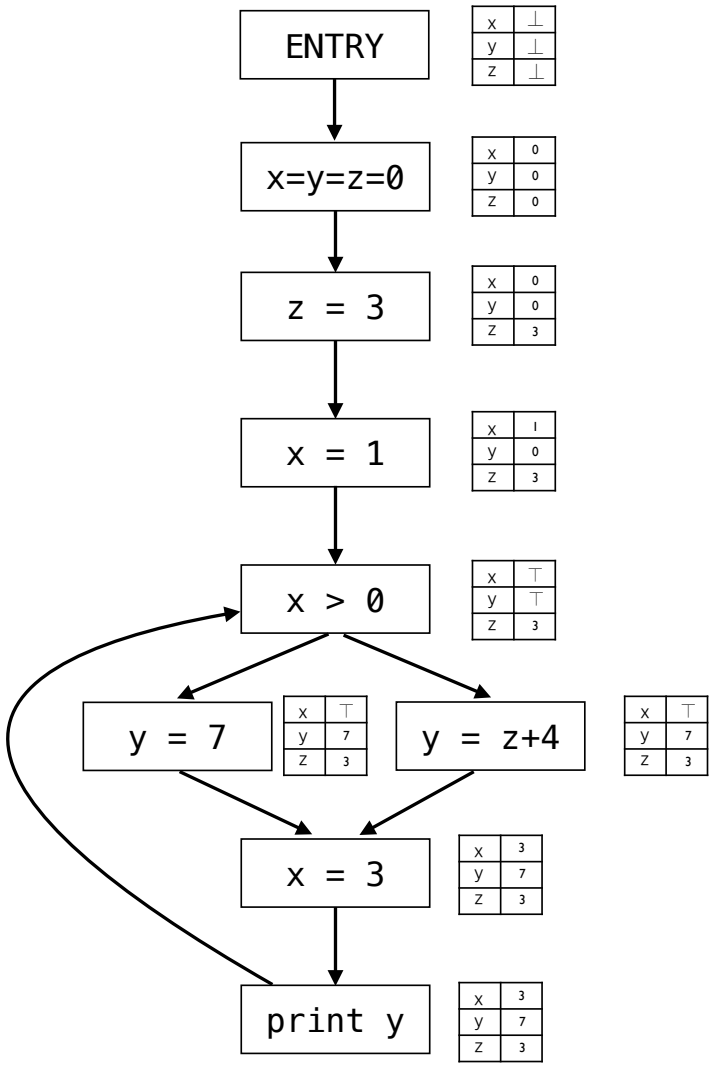
Constant Folding

Decide that the value of an expression is a constant and use it instead.

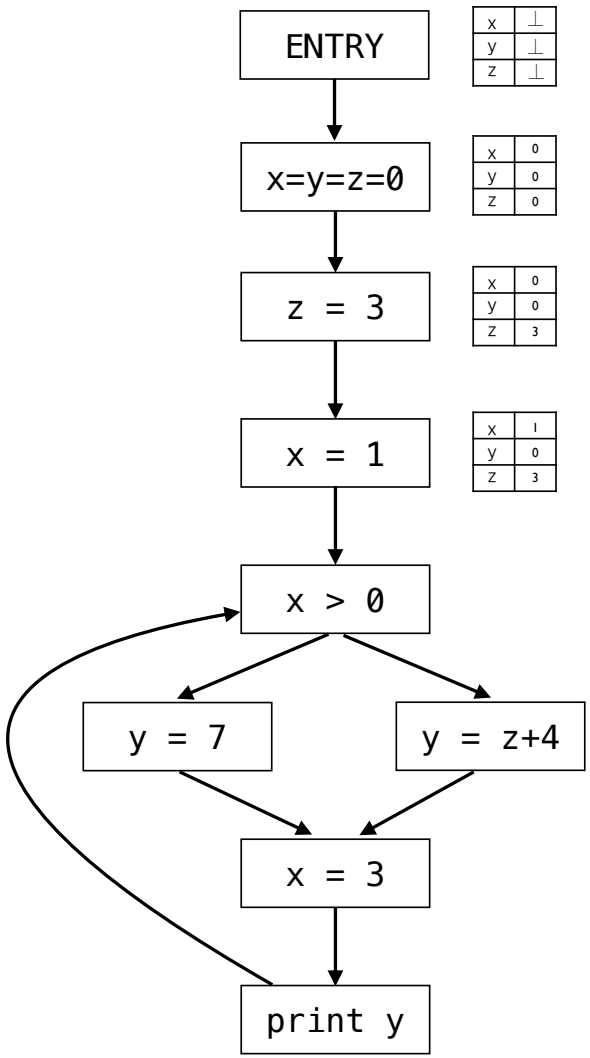


Constant Propagation Analysis

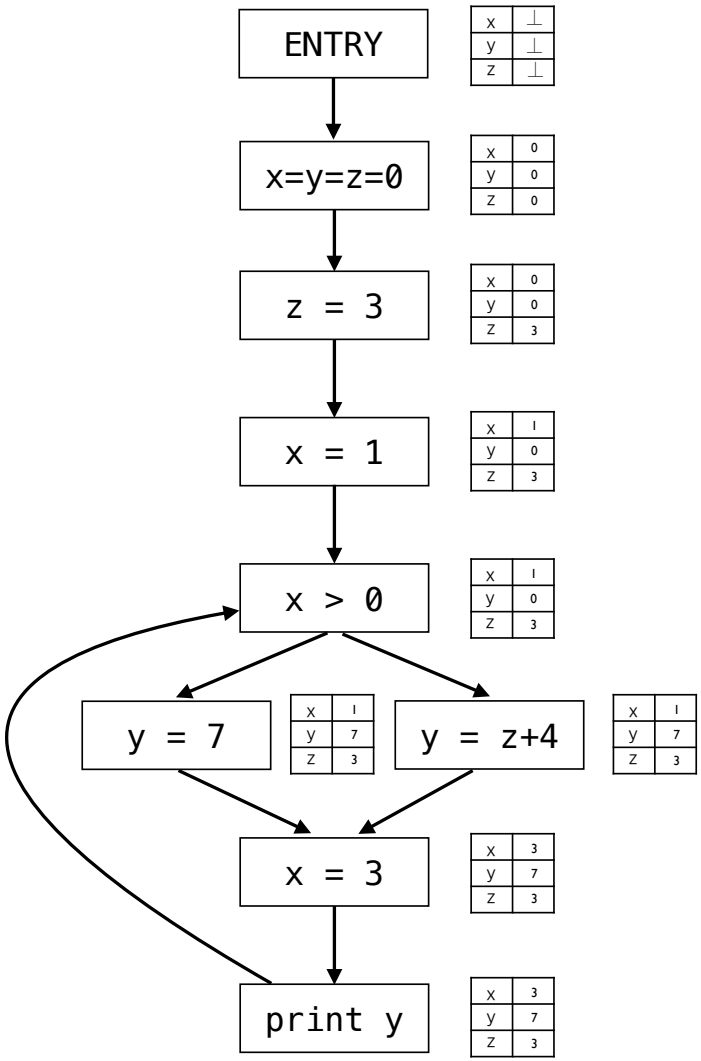
For each program point, determine whether a variable has a constant value whenever execution reaches that point.



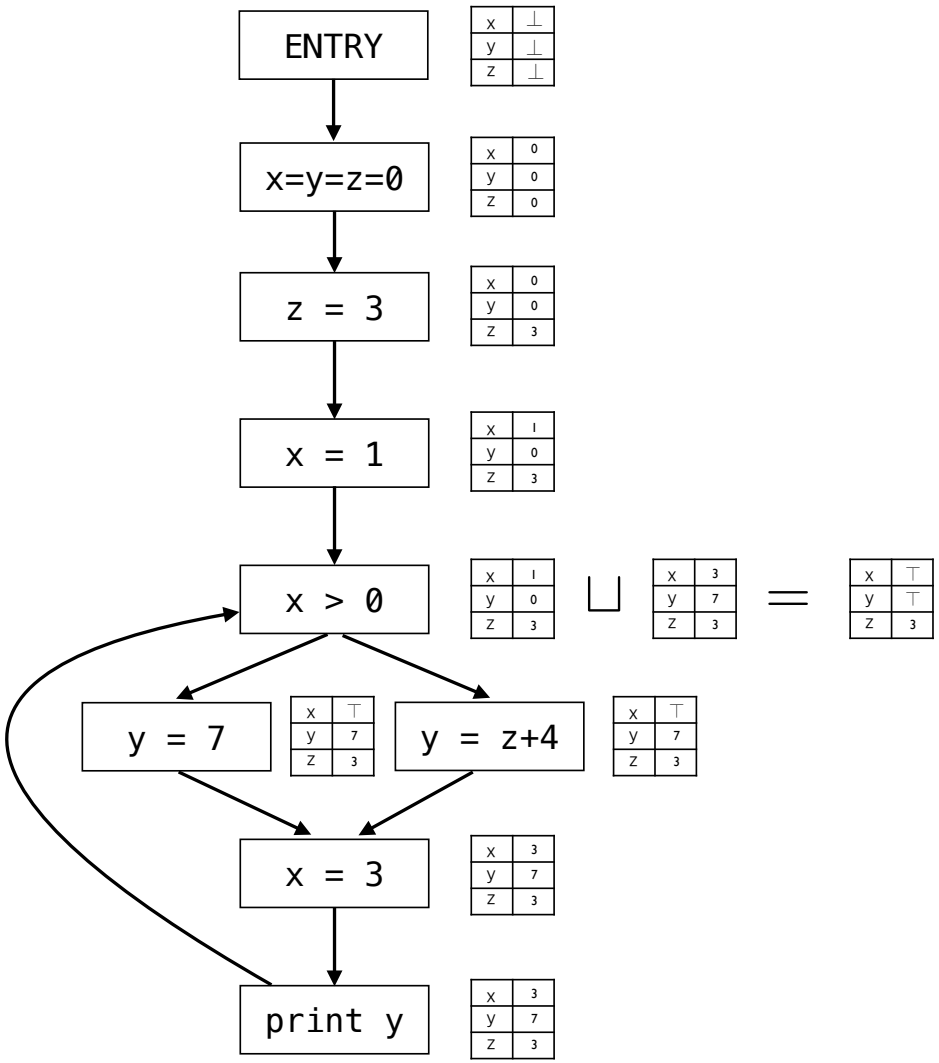
How It Works (1)



How It Works (2)



How It Works (3)

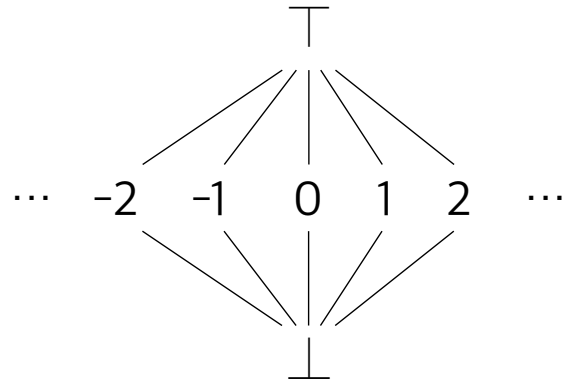


Constant Analysis

The goal is to compute

$$\begin{aligned} \textbf{in} &: \boldsymbol{Block} \to (\boldsymbol{Var} \to \mathbb{C}) \\ \textbf{out} &: \boldsymbol{Block} \to (\boldsymbol{Var} \to \mathbb{C}) \end{aligned}$$

where $\mathbb{C}$ is a partially ordered set:



with the order:

$$\forall \boldsymbol{c_1}, \boldsymbol{c_2} \in \mathbb{C}.\ \boldsymbol{c_1} \sqsubseteq \boldsymbol{c_2} \text{ iff } \boldsymbol{c_1} = \bot \ \vee \ \boldsymbol{c_2} = \top \ \vee \ \boldsymbol{c_1} = \boldsymbol{c_2}$$

Functions in $\boldsymbol{Var} \to \mathbb{C}$ are also partially ordered:

$$\forall \boldsymbol{d_1}, \boldsymbol{d_2} \in (\boldsymbol{Var} \to \mathbb{C}).\ \boldsymbol{d_1} \sqsubseteq \boldsymbol{d_2} \text{ iff } \forall \boldsymbol{x} \in \boldsymbol{Var}.\ \boldsymbol{d_1}(\boldsymbol{x}) \sqsubseteq \boldsymbol{d_2}(\boldsymbol{x})$$

Join (Least Upper Bound)

The *join* between domain elements:

$$c_1 \sqcup c_2 = \begin{cases} c_2 & c_1 = \bot \\ c_1 & c_2 = \bot \\ c_1 & c_1 = c_2 \\ \top & \text{o.w.} \end{cases}$$

The join between abstract states:

$$d_1 \sqcup d_2 = \lambda x \in Var.\ d_1(x) \sqcup d_2(x)$$

Transfer Function

The transfer function

$$f_B : (Var \to \mathbb{C}) \to (Var \to \mathbb{C})$$

models the program execution in terms of the abstract values: e.g.,

- Transfer function for $z = 3$:

$$\lambda d.\ [z \mapsto 3]d$$

- Transfer function for $x > 0$:

$$\lambda d.\ d$$

- Transfer function for $y = z + 4$:

$$\lambda d. \begin{cases} [y \mapsto \bot]d & d(z) = \bot \\ [y \mapsto \top]d & d(z) = \top \\ [y \mapsto d(z) + 4]d & \text{o.w.} \end{cases}$$

Transfer Function

A simple set of commands:

$$\begin{array}{rcl} c & \rightarrow & x := e \mid x > n \mid \\ e & \rightarrow & n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \end{array}$$

The transfer function:

$$\begin{array}{rcl} f_{x:=e}(d) & = & [x \mapsto [\![e]\!](d)]d \\ f_{x>n}(d) & = & d \\ \\ [\![n]\!](d) & = & n \\ [\![x]\!](d) & = & d(x) \\ [\![e_1 + e_2]\!](d) & = & [\![e_1]\!](d) + [\![e_2]\!](d) \\ [\![e_1 - e_2]\!](d) & = & [\![e_1]\!](d) - [\![e_2]\!](d) \end{array}$$

Data-Flow Equations

Equation:

$$\textbf{in}(B) = \bigsqcup_{P \hookrightarrow B} \textbf{out}(P)$$
$$\textbf{out}(B) = f_B(\textbf{in}(B))$$

Fixed point computation:

For all $i, \textbf{in}(B_i) = \textbf{out}(B_i) = \lambda x.\bot$
while (changes to any **in** and **out** occur) {
  For all i, update
    $\textbf{in}(B_i) = \bigsqcup_{P \hookrightarrow B_i} \textbf{out}(P)$
    $\textbf{out}(B_i) = f_{B_i}(\textbf{in}(B_i))$
}

Summary

- Data-flow analyses we covered:
  - Reaching definitions analysis
  - Liveness analysis
  - Available expressions analysis
  - Constant propagation analysis
- Optimization passes
  - Common subexpression elimination
  - Copy propagation
  - Decode elimination
  - Constant folding